A RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment

Gianluca Bellocchi*, Alessandro Capotondi*, Francesco Conti †, Andrea Marongiu*
*University of Modena and Reggio Emilia, Modena, Italy †University of Bologna, Italy
Email: *name.surname@unimore.it †f.conti@unibo.it

***Abstract*—Modern cyber-physical systems (CPS) are increasingly adopting heterogeneous systems-on-chip (HeSoCs) as a computing platform to satisfy the demands of their sophisticated workloads. FPGA-based HeSoCs can reach high performance and energy efficiency at the cost of increased design complexity. High-Level Synthesis (HLS) can ease IP design, but automated tools still lack the maturity to efficiently and easily tackle system-level integration of the many hardware and software blocks included in a modern CPS. We present an innovative hardware *overlay* offering *plug-and-play* integration of HLS-compiled or handcrafted acceleration IPs thanks to a customizable *wrapper* attached to the overlay interconnect and providing shared-memory communication to the overlay cores. The latter are based on the open RISC-V ISA and offer simplified software management of the acceleration IP. Deploying the proposed overlay on a Xilinx ZU9EG shows $\approx$ 20% LUT usage and $\approx 4\times$ speedup compared to program execution on the ARM *host* core.**

I. Introduction

Modern *cyber-physical systems* (CPS) are adopting increasingly high degrees of autonomy in their designs. Increased autonomy requires adequate on-board smart sensing and computing capability to support safe decision making, based on large amounts of data that is sensed, analyzed and understood in real-time. Key to providing such computing power within the tight energy budget of typical CPS is the adoption of high-end embedded systems-on-chip based on multi-core CPU plus highly parallel acceleration logic like GPGPU or FPGA.

FPGA-based heterogeneous systems-on-chip (HeSoCs) are being increasingly adopted in this context, particularly when streaming data sources are involved [1] or when Machine Learning (ML) and Deep Learning (DL) *inference* is concerned [2], [3]. A comparative study of FPGA, GPU, and FPGA+ASIC in-package solutions for persistent DL has shown that FPGAs can offer 2.7$\times$ (FP32) to 8.6$\times$ (INT8) lower latency than GPUs across RNN, GRU, and LSTM workloads from DeepBench [4]. The capability of flexibly defining parallel, non-Von-Neumann processing logic and custom memory hierarchies, all within contained power envelopes, makes the FPGA an ideal candidate for *acceleration*. The main limiting factor in the adoption of FPGAs is the arduous development process. FPGA accelerators often leverage full-custom design flows to achieve maximum performance, using conventional RTL hardware design techniques, and requiring low-level hardware knowledge and a long and complex design process. Coupled with very long compilation times, this often results in significant productivity issues. EDA tool vendors have widely adopted *High-Level Synthesis* (HLS) tools to address this issue. However, while HLS allow designers to focus on high-level functionality instead of low-level details, automated tools still lack the required maturity to efficiently and easily tackle *system-level integration* of the many hardware and software blocks included in a modern CPS.

One alternative technique for improving design productivity is to use a virtual hardware representation that overlays the original FPGA fabric, referred to as an *overlay* architecture [5]. *Overlays* are programmable, coarse-grained hardware abstraction layers on top of FPGA hardware, offering a higher-level programming approach. *Overlays* also mitigate the slow compilation problem by avoiding the complex FPGA design flow – resulting in improved design productivity – and offer the advantage of rapid swapping of architectural blocks, as coarse-grained overlay architectures have smaller configuration data sizes than fine-grained FPGAs.

In this paper, we present an innovative *overlay* that simplifies the adoption of Commercial-off-the-Shelf (COTS), FPGA-based HeSoCs coupling physical *host* CPU and DRAM with programmable logic (PL). Our *overlay* is deployed on the PL of such HeSoCs and leverages soft-cores for flexible control of user-defined, application-specific accelerators. Different accelerators can flexibly operate and re-configure their operation without the costly need for *host* intervention, thus avoiding significant performance degradation. Normal accelerator operation and accelerator reconfiguration can both be achieved via standard computation *offloading* from the *host* CPU to the soft-cores (e.g., OpenMP v4.x+). The user can rely on any methodology of his/her choice to design the accelerators (e.g., HLS); the overlay includes dedicated logic (the *wrapper*) to provide *plug-and-play* HW/SW integration of such accelerators.

Our experimental setup explores an instance of the proposed overlay deployed on a Xilinx ZU9EG MPSoC. The results show that the overlay alone uses $\approx$ 23% LUT, $\approx$ 12% FF and $\approx$ 3.8% BRAM resources. The performance is comparable to heavily hand-optimized Vivado codes (obtained with much higher developer effort compared to our solution) and 4.08$\times$ faster than their ARM host core counterpart.

The rest of this paper is organized as follows: Section II describes related work and introduces the architectural blocks from which we build our proposed architecture. Section III presents our overlay and the methodology for integrating custom-designed accelerators. Section IV discusses our experimental setup and results. Section V concludes the paper.

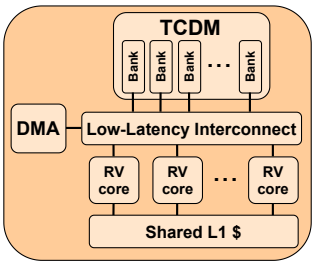

Fig. 1: PULP cluster.

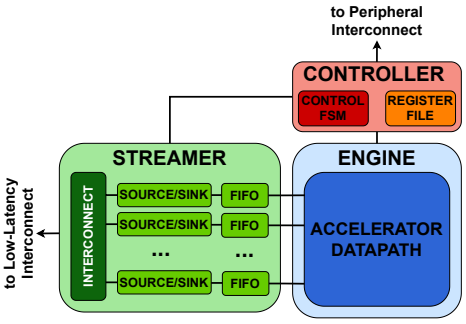

Fig. 2: Template of a HWPE. [6].

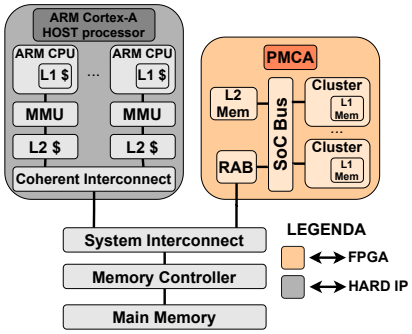

Fig. 3: HERO hardware architecture.

II. Background

A. Related Work

System-level design (SLD) is one of the main challenges to be faced in the construction of CPS, which are composed of multiple hardware and software modules. The accelerator-rich architecture of Chen et al. [7] employs an automated flow with HLS tools, accelerator virtualization and a system-level integration strategy to reduce the engineering cost. Differently from our case, the architecture does not leverage HeSoCs but employs the FPGA for verification purposes, the authors' main goal being to provide guidelines for ASIC implementation.

The overlay architecture proposed by Ng et al. [8] consists of a RISC-V-based soft-core with ISA extensions to implement runtime transfer of control to a tightly-coupled hardware accelerator. Similar to our work, the authors demonstrate that hybrid software-hardware applications achieve comparable performance to pure-hardware implementations. On the contrary, our overlay concept embraces the whole architecture supporting the operation of the accelerators. Mantovani et al. [9], [10] proposed a methodology for heterogeneous SoC design that combines a modular tile-based architecture with a variety of flows for the design and optimization of accelerators. Lomuscio et al. employ a library of hardware kernels that can be loaded on-the-fly on FPGAs in partial reconfigurable PL portions [11]. The design of the kernels is accomplished with tools such as Vivado HLS and wrapped similarly to our proposed flow. The authors exploit an IP-Wrapper to supply the accelerator with an interface to the surrounding environment. Our solution adopts a cluster-based architecture, enabling important hardware optimizations on the hardware kernels in terms of memory bandwidth and computation parallelism.

Other critical aspects in the design of hardware overlays are the target frequency and the memory bandwidth. To cope with these problems, Gray proposes a shared-memory cluster-based architecture [12], including a massively parallel accelerator array based on a RISC-V core. This solution eases system design through software implementations of heterogeneous applications, and a NoC architecture to cope with system bandwidth. Compared to our overlay, the nature of the acceleration resource is different since our overlay exploits hardware workloads and a local soft-core to implement multi-accelerator interaction.

A typical downside of overlay architectures is a conspicuous use of logic resources, resulting in considerable implementation overheads. Interconnection networks and storage elements are typically the most area-hungry components. Li et al. tackle resource overhead proposing a time-multiplexed overlay solution employing a simple linear interconnect [13]. Taras et al. demonstrate that optimizing the overlay infrastructure on the features of the underlying FPGA architecture is another beneficial strategy to face performance and area overheads [14].

Concerning programmability, Coole et al. investigate virtual architectures [15] implemented in between user designs and the physical device to abstract the FPGA fabric. The strategy enhances ease of use, system portability, and reduces compilation times. Again, the main drawback is the area overhead that the authors tackle through optimization techniques targeting specific application domains. Wilson et al. [16] presented an overlay-based infrastructure for FPGA development. Despite some similarities with our design key features, this work focuses mainly on design and implementation timing improvement, with no in-depth assessment of the application-level performance achievable with their solution. We believe our proposed overlay architecture could synergistically co-operate with most of these approaches.

B. The PULP Project and HERO

The Parallel Ultra Low Power (PULP) Platform is an open, scalable HW/SW research and development platform aimed at exploring highly parallel architectures for ultra-low-power processing [17]. PULP includes a state-of-the-art microcontroller system, with efficient 32-bit [18] and 64-bit implementations [19] based on the open-source RISC-V instruction set architecture, and a multi-core platform based on a scalable *cluster* architecture. A PULP *cluster*, depicted in Figure 1, consists of a parametric number of cores compliant to the RV32IMC ISA and enhanced with DSP-like extension for performance boost and code size reduction. The cores in the *cluster* share data through a multi-banked, L1 Tightly-Coupled Data Memory (TCDM) and a low-latency interconnect. Data are moved in and out of the TCDM via a DMA engine. Instruction caching leverages a shared IP to minimize code replication, given the single-program, multiple-data execution model primarily targeted by this type of architecture [20].

Hardware Processing Elements: PULP clusters can be enhanced with application-specific accelerators, called Hardware Processing Elements (HWPE) [21], to deliver higher levels of

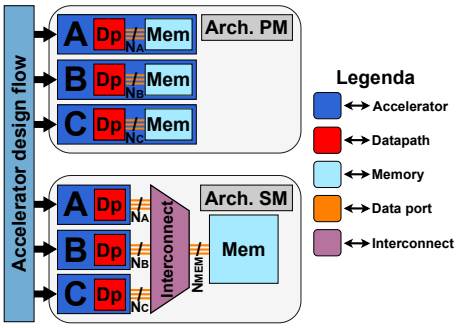

Fig. 4: Private-memory (PM) vs. Shared-memory (SM) Accelerator-based Architecture Models.

performance and energy efficiency for specific tasks. Unlike many accelerator designs HWPEs do not necessarily rely on an external DMA to feed them with input data and extract output data, and they are not tied to a single core. Instead, they operate directly on the same TCDM that is shared by other cores in the *cluster*, with memory-mapped control through a peripheral interconnect. HWPE execution can be readily interspersed with software running on the cores, as all that needs to be exchanged between the two is a set of pointers.

Practical HWPE implementations have been proposed for several applications, including convolutional [22] and binary neural networks [23], but the concept can be generalized. Fig. 2 depicts a generic template for a HWPE. It is composed of three distinct modules: an *engine*, a *streamer* and a *controller*. The *engine* is the accelerator datapath itself; it is entirely application-specific and operates on latency-insensitive streams of data. The *streamer* acts as a translator between the memory-mapped communication protocol of the TCDM and the streaming protocol of the accelerator datapath. To adapt to application-specific bandwidth requirements, on the TCDM side, the *streamer* exposes a configurable number of master ports to the low-latency interconnect. The *controller* allows for HWPE programming via a memory-mapped control interface, with control registers and a finite state machine (FSM) implementing coarse-grained accelerator control/(re)configuration. Intuitively, both the register file and the FSM include parts that are generic and others that are strongly application-specific.

HERO Platform: HERO [24] is an open-source research platform based on FPGA emulation of PULP-based heterogeneous many-core systems. HERO can be instantiated on FPGA SoCs like the Xilinx Zynq family. As shown in Figure 3, HERO combines the physical ARM Cortex-A *host* processor on the SoC with a PULP-based many-core accelerator (PMCA) deployed as a soft-IP on the FPGA. The *host* processor is typically a multi-core CPU capable of running unmodified Linux. The PMCA consists of one or more PULP *clusters* interconnected via a NoC. At the top level, the clusters share a multi-banked L2 scratchpad memory (SPM) and a *remapping address block* (RAB), a simplified IOMMU [25], [26].

C. Motivation

Production-level HLS methodologies, like those offered by Xilinx design tools, are pretty mature in terms of single-IP development. The same does not fully hold for system-level integration of multiple IPs [27]. To demonstrate the potential our overlay exhibits to tackle system-level *optimization* and *integration* of multiple HWPEs we discuss and examine: (i) a typical HLS flow, as offered by modern tools; (ii) how different architectural models relate to SLD.

User Perspective on a Standard HLS Flow: HLS flows start from a high-level specification (C, C++, or SystemC) of an accelerator, which is then compiled into custom RTL. Several optimization knobs exist to improve the accelerator performance, but their reach does not extend to *inter-process specifications*, thus making the design flow agnostic of the whole system structure [28]. IP packaging is based on the IP-XACT standard to ease *system-level integration*, that is carried out at the block-level. Following this methodology, the user is asked to create the *architecture* surrounding the HLS-compiled IP. Interconnecting the latter with other system IPs is accomplished through customizable AXI-compliant interconnects. Multiple HLS IPs are typically composed through AXI4-Stream interfaces, based on a lightweight protocol for unidirectional data streams. MicroBlaze soft-cores can be employed to implement complex control schemes, but customization is not feasible due to their closed-source nature. Concerning the design of the *memory* subsystem, although Block RAMs (BRAMs) are distributed across the FPGA fabric to optimize memory access from any arbitrary mapping of the software data structures, memory ports are limited: shared multi-bank memories are key for scalability and to avoid the extensive memory block replication implied by naive IP-centric designs. Accomplishing this task in standard Xilinx tool-flows is complicated by the IP-centric view of the approach.

Accelerator Optimization Strategies: A generic, IP-centric design flow typically interacts with either private-memory (PM) or shared-memory (SM) architectures, as shown in Figure 4. Considering a set of different accelerators (A, B, and C), the designer can optimize the memory subsystem in different ways [29]. Typical of IP-centric methodologies, the PM architecture exploits private, multi-port memories to increase the concurrent W/R operations, thus improving design throughput. To this end, N_* data ports have to be routed from the memory banks to the datapath, as shown in Figure 4 for the PM architecture. However, in the context of accelerator-rich architectures IP-centric optimization of the memory subsystem quickly eats out large portions of the accelerator area [30], reducing the number of accelerators that can be hosted on the programmable logic. Lastly, the way accelerators interact with memory is of primary importance when it comes to SLD [31]. According to our experience, it is not always so straightforward to enforce the desired data management policy in hardware. A typical example is *double-buffering*, which is not so easy to be mapped to complex computation schemes, especially if the user wants more datapaths to co-operate on the same buffer. In these cases, it is easier to implement such control schemes employing software routines.

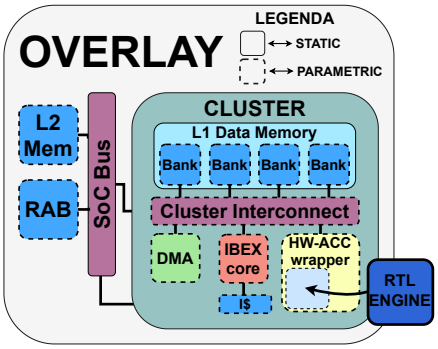

Fig. 5: Architecture of the proposed FPGA hardware overlay.

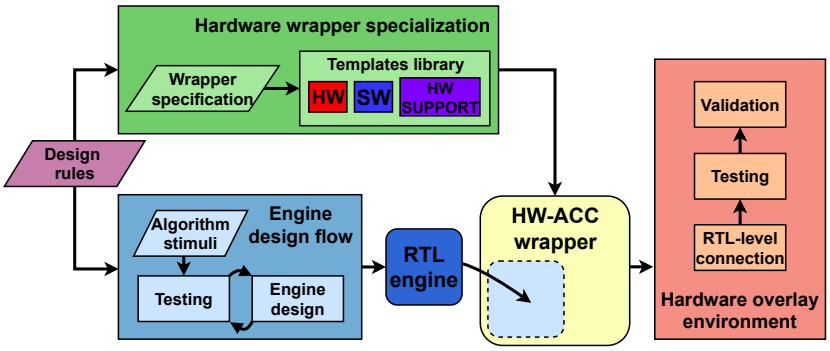

Fig. 6: Design flowchart: from the design rules up to the deployment of application-specific accelerators.

III. Overlay Architecture

Compared to the standard flow that exploits IP-private memories, a multi-bank, low-latency shared memory greatly improves area utilization, provided that the interconnection system is carefully designed [7] [13] [14]. As shown in Figure 4 for the SM architecture, this specification implements N_* data ports on each accelerator interface to feed their parallel datapaths (N_A ports for accelerator A, etc.). Designing this interface with Xilinx Vivado HLS can be achieved in a non-straightforward manner by exploiting interleaved array partitioning on streaming interfaces (typically employed for accelerator design), but this is usually beyond the expertise of the typical HLS user. Such architectural details should be rather abstracted by the *overlay* architecture, along with their careful design and implementation.

HERO constitutes a convenient starting point to implement such *overlay*: being conceived as a many-core architecture, HERO naturally complies with some of the basic requirements to build an accelerator-rich design, most notably the cluster-based design and the multi-bank shared memory design. However, HERO *clusters* are designed for general-purpose (or, at best, signal-processing oriented) parallel execution and thus have substantial limitations in the context of FPGA hardware acceleration that we target. HERO uses the FPGA merely as a medium for emulation of projects meant for IC realization. The proposed *overlay* uses the FPGA as a target for acceleration. For an overlay to be an efficient and convenient solution, it should offer: (i) SLD capabilities; (ii) transparent accelerator integration flow; (iii) streamlined resource usage. In the following, we discuss how we redesign the PULP *clusters* to achieve these goals and to enable plug-and-play integration of custom accelerator logic in our overlay.

A. Overlay Architecture

Figure 5 shows an overview of the proposed *overlay*. At the *cluster* level, there are a few key modifications that we make. First, the cores in our proposal are not meant to serve as *data crunchers*, but rather as flexible control cores. For this reason, (i) we need far fewer cores than in the original design; (ii) we need the cores to be much simpler. A single core might suffice to orchestrate the operation of the accelerators and the data movements. However, we believe a second one might be useful when small application code parts are better executed close to the accelerator than on the *host* cores. Moreover, since the proposed overlay can be programmed with OpenMP, one core could be in charge of executing OpenMP runtime library code. At the same time, the other is always available to control the accelerators or execute application code. Concerning the simplicity of the design, we replace the original RISCY core with an IBEX core [32], an area-optimized 32-bit RISC-V core implementing the RV32IMC instruction set architecture (ISA). IBEX exhibits two pipeline stages and a simpler ALU compared to RISCY.

Since the cores in our *overlay* are mostly meant for *Multiple-Program, Multiple-Data* (MPMD) execution, the shared instruction cache (I-cache) has been replaced with a private one, which has more straightforward control logic. Atomic Memory Operation (AMO) support modules have been removed since parallel execution on the soft-cores is not a desired feature.

The reduction in the number of cores also implies a reduction in the number of master ports on the TCDM interconnect, that can be devoted to the accelerators instead. The size of the TCDM, the number of banks or the DMA burst size are parameters in our design to be adapted to the application's specific requirements at hand. Figure 5 shows blocks that can be parameterized as dashed boxes.

The most significant modification that we propose at the *cluster* level is the introduction of a parametric *wrapper*, which is meant to provide *plug-and-play* integration of accelerators in the platform. This is discussed in Section III-B.

At the top level, other customizable IPs are the RAB and the L2 memory. The RAB can be removed entirely – in case shared virtual memory is not relevant for the target system – or reconfigured in terms of the number of TLB entries and overall size. The L2 can be configured in size and number of banks.

B. Accelerator Integration Methodology

The integration of custom accelerators is simplified by the definition of a communication/control interface in the form of a *wrapper*. This encapsulates the functionality of an HWPE *streamer* and *controller* modules; since both exhibit dependencies on the *engine* implementation, it is necessary to distinguish between their static and variable RTL components. We stress

that our proposal does not mandate any specific method to design accelerator engines. Both HLS tools or manually optimized HDL code are supported. Once the user has defined the fundamental properties to support its engine, the wrapper specialization is performed via *template* parameterization.

Figure 6 shows an overview of the complete integration flow. Starting from a set of design rules, the designer is equipped with a high-level language (HLL) interface to allow *wrapper* customization. An automated flow allows for the integration of the engine logic within the *wrapper*, which is in turn connected to the *overlay* for validation.

The following sections describe the flow through an exemplary integration of a Matrix Multiplication engine.

1) Wrapper Specialization: We use the Mako[1] Python library for defining a list of RTL templates for the *wrapper* hardware components. Hence, the integration procedure is abstracted since the designer can modify the attributes of a Python class and spread them through the entire wrapper template library. Listing 1 shows an example of the attributes of this class, that may either regard the *engine*, the streaming interface, and both the standard and application-specific registers from the register file: this includes the target HWPE name; the list and number of streams incoming to/outgoing from the *engine*; and the number and nature of standard and accelerator-specific (custom) registers. These are only a subset of the knobs that are available in the customization tool. These specifications are propagated throughout the wrapper template library generating: (i) System Verilog templates for all components of the *wrapper*; (ii) Interface to connect with the HLS-compiled or handcrafted engine; (iii) Software library for HWPE runtime calls.

```
class hwpe_specs:
    def __init__(self):
        self.hwpe_target         = 'mmult'
        self.list_sink_stream    = [ 'in1' , 'in2' ]
        self.list_source_stream  = [ 'out' ]
        self.n_sink              = len(self.list_sink_stream)
        self.n_source            = len(self.list_source_stream)
        self.std_reg_num         = 5
        self.custom_reg_name     = [ 'custom_reg' ]
        self.custom_reg_dim      = [ 32 ]
        self.custom_reg_num      = len(self.custom_reg_name)
```

Listing 1: Python interface for wrapper specialization.

```
1  void mmult(data_t in1[Win], data_t in2[Win], data_t &out, data_t custom_reg )
2  {
3  #pragma HLS INTERFACE axis port=in1
4  #pragma HLS INTERFACE axis port=in2
5  #pragma HLS INTERFACE axis port=out
6
7  #pragma HLS array_partition variable=in1 cyclic factor=Np
8  #pragma HLS array_partition variable=in2 cyclic factor=Np
9
10     const unsigned mat_dim = MATRIX_WIDTH;
11     data_t result = 0;
12
13     data_t local_reg_value = custom_reg;
14
15     engine_loop_A: for(int i = 0; i < mat_size; i+=Np){
16     #pragma HLS LOOP_TRIPCOUNT min=Wp max=Wp
17     #pragma HLS PIPELINE
18         engine_loop_B: for(int j = 0; j < Np; j++){
19         #pragma HLS LOOP_TRIPCOUNT min=Np max=Np
20             result += in1[i+j] * in2[i+j];
21         }
22     }
23     out = result;
24 }
```

Listing 2: C++ design of a Matrix Multiplication engine.

[1]https://www.makotemplates.org/

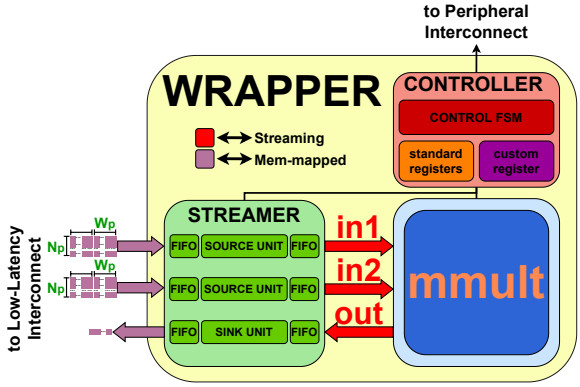

Fig. 7: Example of integration of a HLS-compiled Matrix Multiplication engine inside an *overlay wrapper*.

2) Engine Interface: For the integration to be successful, the engine design flow should employ an interface protocol consistent with the one the wrapper adopts. The interface is designed to be compatible with Vivado HLS but can also be easily connected to a handcrafted *engine*. A set of four *control signals* are used to notify the *wrapper* of the *engine* running state and to modify it: `start` notifies the engine that it can start processing; `done` notifies the wrapper that the engine terminated its operation; `idle` is 1 if the engine is not active, and `ready` is 1 if the engine is ready to accept inputs.

Beyond this small set of control signals, the other engine input and output signals are fully application-specific. The first communication class is that of direct inputs and outputs of the datapath, expressed as streams: unidirectional data-flows without an attached address designed to be latency-insensitive with well-defined handshaking procedures. The aforementioned lightweight data transfer solution is ideal for FPGA-based accelerators. We support two similar protocols AXI4-Stream[2] (used by Vivado HLS) and HWPE-Stream [6]. The number of streaming ports can be controlled through *ad-hoc* wrapper customization as explained in Section III-B1.

The *wrapper* comprises a configurable number of memory-mapped custom registers. Before the datapath starts its computation, these registers are written by the controlling RISC-V core through the wrapper slave port.

3) HLS Engine Design: Listing 2 shows the HLS specification for a Matrix Multiplication engine. The engine top module is declared at line 1. Its name is the same specified in the *hwpe_target* attribute of Listing 1. The *pragma* directives of lines 3-5 implement I/O arrays with an AXI4-Stream protocol. A static algorithm-specific parameter *custom_reg* is declared at line 1 to be implemented as a single data port at RTL for contexts of continuous use during the engine's operation. The algorithm operations are specified at lines 15-22.

As explained in Section II-C, to optimize the code are various strategies. The specification at line 17 induces *spatial parallelism* through pipelining of `engine_loop_A`. This automatically unrolls `engine_loop_B` with a parallelism factor of N_P. This way, for the engine to output a result, a total

[2]https://www.xilinx.com/products/intellectual-property/axi.html

TABLE I: Xilinx ZU9EG Resources.

	Availability
LUT (K)	274
FF (K)	548
BRAM (Mb)	32.1
DSP (Slices)	2520

TABLE II: Out-of-cluster IPs and resource utilization.

	LUT utilization (%)	FF utilization (%)
AXI data converters	3.35	0.33
AXI read burst buffers	4.44	6.43
SoC bus	1.92	0.93
RAB	6.11	1.85
L2 memory	0.30	0.10

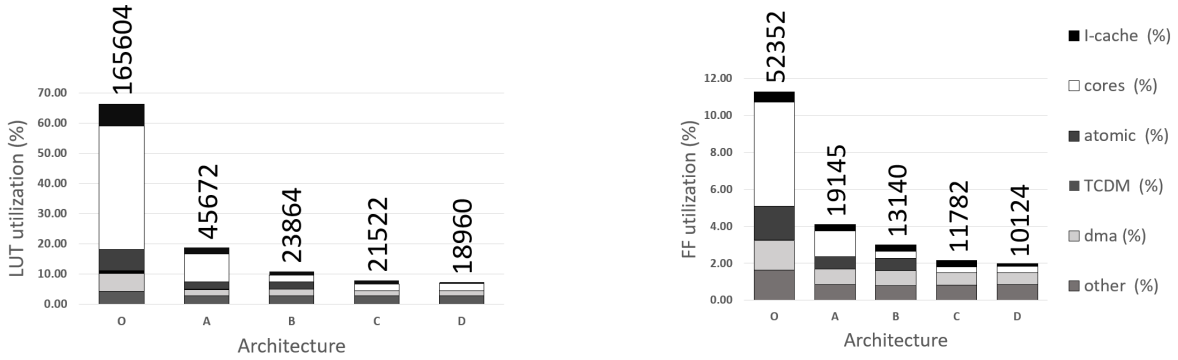

Fig. 8: LUT and FF utilization breakdown for the different architecture presented in Section IV-A.

of W_P data are expected, where $W_m = W_P * N_P$. To fully take advantage of latter optimization, input interface arrays *in1* and *in2* are partitioned into N_P sub-arrays at lines 7-8. Our flow supports this optimization by automatically specializing both the *wrapper* and *overlay* with parallelized source units and additional master ports on the *overlay* interconnect. At the same time, TCDM banks can be increased to limit *memory contention behaviour*. *Port interleaving* is also exploitable with proper wrapper programming.

Figure 7 shows the resulting engine integration.

4) Software Integration and Execution Model: Applications initially run on the *host* processor that is in charge of allocating the input data buffers in a dedicated non-cached, non-paged DRAM portion that is accessible by the *overlay* system. The *offloading* procedure is based on memory copy operations.

At this point, the *overlay cluster* is activated, and the *overlay core* starts executing, transferring data blocks from DRAM (L3) to TCDM (L1). If the dataset is larger than the L1 size (128kB), the soft-core partitions data into smaller blocks.

The development of the data transfer routine is not transparent to programmers to offer them free choice of implementation. *Double-buffering* is an example of an implementable control routine to reach the performance entry together with hardware optimizations pertinent to the HLS/HDL design of the hardware accelerator. Other than transferring data, the soft-core can process them (not its final goal, though) or control the accelerator operation. *Accelerator programming* is essentially divided into two phases. At first, the accelerator is initialized, and the address generator is programmed. To clarify, since the memory interface of the wrapper (namely, the streamer) translates memory accesses into data streams (and vice versa), it needs to be programmed according to the pattern of the L1 accesses, which is algorithm-dependent. For this reason, we have extended the *wrapper customization flow* inferring a set of registers per input and output port to offer a generalized and practical way of programming the address generator. This is a *one-time cost* as soon as accelerators employ a single working context. Besides, if a double-buffered solution is engaged, the wrapper controller needs to be updated with the pointer to the data to be crunched; this represents the second and *recurrent cost* associated with the accelerator programming. Besides, *accelerator execution* consists of the writing to a wrapper register to trigger execution and the waiting interval where the core waits for the generation of an end-of-computation event from the wrapper itself.

IV. Experimental Results

In this section, we evaluate our overlay in terms of resource utilization and performance. The proposed *overlay* has been synthesized and deployed on a Xilinx Zynq UltraScale+ ZU9EG MPSoC. We used Xilinx Vivado HLS v2018.2 to design accelerated *engine*, Xilinx Vivado v2019.2 to synthesize and implement RTL designs, Xilinx Petalinux v2019.2 to deploy the application on the target board.

A. Overlay Resource Utilization

Table I shows the available LUTs, FFs, BRAMs and DSP slices in the PL of tested MPSoC. As a first experiment, we evaluate the relative (and absolute) use of resources implied by our *overlay* measuring the resource utilization by those components that lie outside of the *cluster*, which we don't change in the remainder of this section. Table II reports measured values and shows that these components occupy well below 10% of the whole available resources. Focusing on the cluster IPs, we consider five different architecture variants to show the impact of various blocks on resource utilization. Hence, Figure 8 shows breakdowns of LUT (left side) and FF (right-side) usage for the cluster IPs for different architectures. We discuss the usage of BRAM and DSP slices in the text since only a few IPs are concerned. *Architecture O* refers to the original HERO cluster with 8 RISCY cores and only serves as

a baseline. *Architecture A* reduces the number of RISCY cores to 2; *Architecture B* replaces RISCY cores with IBEX cores; *Architecture C* removes atomic instructions; *Architecture D* replaces the shared Instruction Cache with private ones.

The breakdown bars of Figure 8 show resource usage for *I-cache* (shared or private), *atomic* memory operations (AMO), *TCDM* (TCDM banks + logarithmic interconnect) and *DMA*, while other minor IPs are collected in *other*. Reducing the number of RISCY cores ($Archit_{O\rightarrow A}$) impacts the occupation of the shared I-cache (BRAM usage decreases by 26.31%), the DMA and the AMO modules. LUTs, FFs and DSPs decrease respectively by 43.76%, 6.06%, and 2.14%. Replacing the RISCY cores by the IBEX ones ($Archit_{A\rightarrow B}$) additionally reduce occupation: -7.96% LUTs, -1.10% FFs, -3.51% BRAM, and -0.63% DSPs. AMO removal ($Archit_{B\rightarrow C}$) reduces LUT by 2.48% and FF by 1.10%, while employing a private I-cache($Archit_{C\rightarrow D}$) reduces BRAM usage by 5.81%.

To sum up, the actual implementation of our *overlay cluster*, hence *Architecture D* together with the components reported in Table II, results in the following occupation: LUT $\approx$ 23%, FF $\approx$ 12%, BRAM $\approx$ 3.8% and DSP $\approx$ 0%.

B. Overlay-Based Accelerators Profiling and Comparison

The second set of experiments concerns the application level performances enabled by the proposed overlay. We use a Matrix Multiplication (AB) kernel as a benchmark with a fixed matrix size of 512×512 and 32-bit unsigned data elements. The code for a basic and a blocked implementation of the Matrix Multiplication benchmark is provided respectively by the Listing 3 and Listing 4.

```
1
2  void mmult(data_t* in1, data_t* in2, data_t* out, int Wm)
3  {
4    data_t result=0;
5    loop_A: for (int i = 0; i < Wm; i++){
6      loop_B: for (int j = 0; j < Wm; j++){
7        loop_C: for (int k = 0; k < Wm; k++){
8          result += in1[i* Wm + k] * in2[j * Wm + k];
9        }
10       out[i * Wm + j] = result;
11       result=0;
12     }
13   }
14 }
```

Listing 3: Matrix Multiplication algorithm.

```
1
2  void mmult(data_t* in1, data_t* in2, data_t* out, int Wm, int Hs)
3  {
4    data_t result=0;
5    loop_A: for (int ii = 0; ii < Wm; ii+=Hs){
6      DMA_in(in1, local_in1);
7      loop_B: for (int jj = 0; jj < Wm; jj+=Hs){
8        DMA_in(in2, local_in2);
9        loop_C: for (int i = 0; i < Hs; i++){
10         loop_D: for (int j = 0; j < Hs; j++){
11           loop_E: for (int k = 0; k < Wm; k++){
12             result += local_in1[i * Wm + k] * local_in2[j * Wm + k];
13           }
14           local_out[i * Wm + j] = result;
15           result=0;
16         }
17       }
18     }
19     DMA_out(out, local_out);
20   }
21 }
```

Listing 4: Blocked Matrix Multiplication.

1) Overlay Designs: We have implemented overlay-based designs with the *automatic flow* described in Section III-B. These tests run at a target frequency ($f_{overlay} = 90MHz$) and are shortly described in Table III. Speaking of the designs

TABLE III: Designs based on our proposed methodology.

Acronym	Description
O-SW	Pure-software (SW) implementation of Listing 4. Soft-core implements a single-buffered control routine and process data.
O-SW-DB	Pure-software implementation of Listing 4. Soft-core implements a double-buffered (DB) control routine and process data.
O-HW	Hybrid hardware/software implementation of Listing 4. Hardware (HW) acceleration is performed with no datapath optimizations. Soft-core implements a double-buffered control routine.
O-HW-PM-Cpy	Hybrid hardware/software implementation of Listing 4. Datapath is optimized employing partial array partitioning and loop unrolling to implement spatial parallelism (PM). Soft-core implements a double-buffered control routine.
O-HW-PM	Hybrid hardware/software implementation of Listing 4. Datapath is optimized as in Listing 2. Soft-core implements a double-buffered control routine.
O-HW-PM (Proj)	Projection of O-HW-PM at $f_{scaled} = 150MHz$.

of Table III, O-SW is a baseline. During *Loop_A* the soft-core programs the DMA to transfer an $in1$ data block from L3 to L1, then waiting for its completion. *Loop_B* performs the same for $in2$. *Loop_C* and *Loop_D* implement the local buffers addressing for the core to read data lines. The latter are processes in *Loop_E* to generate an output matrix point. Results are buffered in L1 and later transferred back to L3. O-SW-DB adds on top of *Loop_A* and *Loop_B* a double-buffered implementation. In O-HW the algorithm kernel is no more executed by the soft-core, but, instead, an HLS-compiled matrix multiplication engine is wrapped and integrated into the overlay, following the same approach described in Section III-B. Engine design consists of *Loop_E* of Listing 4. Additionally, *Loop_C* and *Loop_D* are implemented in hardware exploiting the wrapper address generator. In O-HW-PM-Cpy, the overlay integrates an HLS-compiled engine containing a prefetching stage, a write stage and private BRAM buffers. Differently from O-HW, *Loop_C* and *Loop_D* are used for local BRAMs addressing. *Loop_E* implements line processing, as for the precedent solution. As explained in Section III-A, our overlay already comprises the required system IPs to support hardware acceleration. Hence, we believe most of O-HW-PM-Cpy components to be in excess. However, this example shows our flow to be flexible to particular user requirements. Finally, O-HW-PM integrates the same engine as of Listing 2 with $N_P = 16$. The wrapped engine logically consists of *Loop_E*, but the compiled RTL description implements N_P loop copies that occur in parallel. Compared to O-HW-PM-Cpy, no additional private memory or read/write stage are included.

2) Overlay Latency Analysis: Figure 9 shows the execution time breakdown of the five different versions of the *overlay-based* matrix multiplication.

Double-buffered copies of *Loop_A* and *Loop_B* tiles allow to

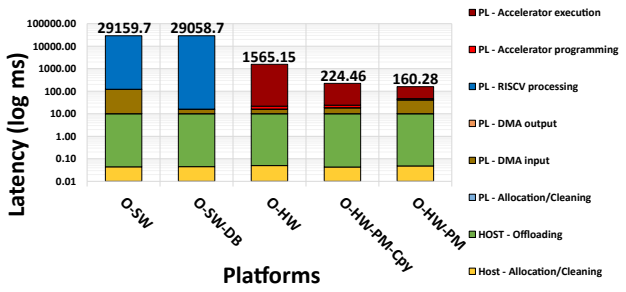

Fig. 9: Overlay application breakdown (Section IV-B2).

TABLE IV: Reference designs based on Xilinx design flows

Acronym	Description
X-HW	Standalone pure-hardware (HW) implementation of Listing 3. No further optimization is specified.
X-BL	Hybrid hardware/software implementation of Listing 4. Data blocking (BL) is implemented.
X-BL-PM	Hybrid hardware/software implementation of Listing 4. Datapath is optimized employing partial array partitioning and loop unrolling to implement spatial parallelism (PM).
X-LOOP	Standalone pure-hardware implementation of Listing 4. Hardware loops (LOOP) are implemented.
X-PM	Standalone pure-hardware implementation of Listing 3. Datapath is optimized employing total array partitioning and loop unrolling to further increase parallelism (PM).

reduce by 18.8× the visible cost of DMA transfers compared to O-SW, as most of it gets hidden by the processing stage (the variation of *PL - RISC-V processing* is negligible). O-HW accelerates kernel execution, thus *PL - RISC-V processing* transforms in *PL - Accelerator execution*. Processing performance improves by a factor of 18.5×, since O-HW implements the whole *Loop_E* in hardware. *Loop_C* and *Loop_D* are hardwired too, resulting in a small additional cost (491672 cycles, including both the one-time and recurrent costs) associated with the wrapper programming. The processing of a single matrix point thus requires 512 accumulations and 1 store cycles to occur executed at $f_{overlay} = 90MHz$. The additional parallelism in O-HW-PM-Cpy reduces *PL - Accelerator execution* by 7× compared to O-HW. O-HW-PM show that the applied optimization permits to further improve the accelerator execution performance, halving the perceived latency. At the same time, the time spent waiting for the DMA to transfer input data blocks is increased, indicating that the DMA input bandwidth has reached saturation.

3) Comparison with Reference Designs: Figure 10 compares our solution with a reference one implemented using the standard Vivado HLS design flow. Each bar is expressed as the superposition of two-timing contributes to the execution time on the host and the FPGA accelerator. We have included a pure-sw implementation of Listing 3 on the *host processor* (H). The former runs at $f_{host} = 1.2GHz$. Besides, are included the

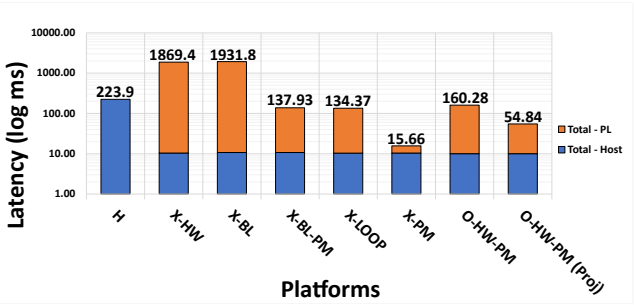

Fig. 10: Platform comparison (Section IV-B3).

execution results concerning the reference designs described in Table IV. Each bar shows the breakdown of two distinct timing contributions: *Total - PL* and *Total - Host*, respectively the measured latency of the portion of application executed on the PL and host processor. *Total - Host* is a common offset of about 10 ms for reference (X-*) and overlay (O-*) designs. The lack of datapath optimizations and the costly host control worsens X-BL performance compared to X-HW by 62.4 ms. *H* implementation runs 8.35× faster than X-HW since it exploits local cache memory to fasten data access. X-BL-PM and X-LOOP feature additional parallelism, which allow them to perform better than *H* by 1.62× and 1.67×, respectively. X-PM is shown for completeness, but in our opinion it is not representative of a practical solution for real-world applications. X-PM copies entires data structures in local memory, achieving the highest performance improvement (14.3× faster than *H*) at the prohibitive expense of $\approx 81.7\%$ BRAM utilization. For larger data structures this approach becomes quickly unfeasible. Concerning our overlay-based solutions, We report results for the O-HW-PM scheme running at 90 MHz, plus a projection (O-HW-PM (Proj)) that considers a target frequency of $f_{scaled} = 150MHz$. The latter is an estimate of the performance of a faster instance of the overlay, that has been obtained linearly scaling the measured breakdown contributes. These allow a performance increase of 1.4× and 4.08×, respectively, compared to *H*.

V. Final Remarks

In this paper we have presented an innovative *overlay* that aims to simplify the adoption of COTS, FPGA-based HeSoCs. By efficiently abstracting the FPGA hardware details, our overlay permits simplifying the deployment of application-specific accelerators and the programmability of the resulting HW/SW platform, thanks to a simple methodology that has been fully described through a practical example of an HLS-compiled engine. Experimental results on a Xilinx ZU9EG MPSoC show $\approx 20\%$ LUT usage and comparable latency to what is achieved with standard Vivado HLS design methodologies (up to 4.08× speedup compared to program execution on the ARM *host* core), with much less developer effort.

We believe our solution is a valuable option for most COTS platforms, particularly if combined with kernel re-configuration techniques, permitting us to accelerate a pool of compute kernels with a transparent user experience. Furthermore, our

roadmap is to improve this solution's resource overhead further, making it appealing for resource-constrained SoC systems.

VI. Acknowledgments

Authors would like to thank the EU commission for funding the ECSEL-JU COMP4DRONES project (No. 826610).

References

[1] M. Quigley, K. Mohta, S. S. Shivakumar, M. Watterson, Y. Mulgaonkar, M. Arguedas, K. Sun, S. Liu, B. Pfrommer, V. Kumar, and C. J. Taylor, "The open vision computer: An integrated sensing and compute system for mobile robots," version: 1. [Online]. Available: http://arxiv.org/abs/1809.07674

[2] X. Liu, H.-A. Ounifi, A. Gherbi, W. Li, and M. Cheriet, "A hybrid GPU-FPGA based design methodology for enhancing machine learning applications performance," vol. 11, no. 6, pp. 2309–2323. [Online]. Available: https://doi.org/10.1007/s12652-019-01357-4

[3] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini, "Neuraghe: exploiting cpu-fpga synergies for efficient and flexible cnn inference acceleration on zynq socs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–24, 2018.

[4] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu, "Why compete when you can work together: FPGA-ASIC integration for persistent RNNs," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 199–207, ISSN: 2576-2621.

[5] X. Li, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "A time-multiplexed fpga overlay with linear interconnect," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1075–1080.

[6] Hardware processing engines — hardware processing engines - interface specifications 1.4 documentation. [Online]. Available: https://hwpe-doc.readthedocs.io/en/latest/index.html

[7] Y.-T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich CMPs: From concept to real hardware," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE, pp. 169–176, event-place: Asheville, NC, USA. [Online]. Available: http://ieeexplore.ieee.org/document/6657039/

[8] H.-C. Ng, C. Liu, and H. K.-H. So, "A soft processor overlay with tightly-coupled FPGA accelerator." [Online]. Available: http://arxiv.org/abs/1606.06483

[9] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E. G. Cota, M. Petracca, C. Pilato, and L. P. Carloni, "Agile soc development with open esp," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[10] P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "High-level synthesis of accelerators in embedded scalable platforms," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 204–211.

[11] A. Lomuscio, G. C. Cardarilli, A. Nannarelli, and M. Re, "A hardware framework for on-chip FPGA acceleration," in *2016 International Symposium on Integrated Circuits (ISIC)*, pp. 1–4.

[12] J. Gray, "GRVI phalanx: A massively parallel RISC-v FPGA accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 17–20.

[13] X. Li, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "A time-multiplexed FPGA overlay with linear interconnect," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1075–1080.

[14] I. Taras and J. H. Anderson, "Impact of FPGA architecture on area and performance of CGRA overlays," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 87–95, ISSN: 2576-2621.

[15] J. Coole and G. Stitt, "Intermediate fabrics: virtual architectures for circuit portability and fast placement and routing," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, ser. CODES/ISSS '10. Association for Computing Machinery, pp. 13–22. [Online]. Available: https://doi.org/10.1145/1878961.1878966

[16] D. Wilson and G. Stitt, "Seiba: An fpga overlay-based approach to rapid application development," in *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.

[17] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "Pulp: A parallel ultra low power platform for next generation iot applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–39.

[18] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-v core with DSP extensions for scalable IoT endpoint devices," vol. 25, no. 10, pp. 2700–2713.

[19] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-v core in 22-nm FDSOI technology," vol. 27, no. 11, pp. 2629–2640.

[20] I. Loi, A. Capotondi, D. Rossi, A. Marongiu, and L. Benini, "The quest for energy-efficient i$ design in ultra-low-power clustered many-cores," vol. 4, no. 2, pp. 99–112.

[21] F. Conti, A. Marongiu, C. Pilkington, and L. Benini, "He-p2012: Performance and energy exploration of architecturally heterogeneous many-cores," vol. 85, no. 3, pp. 325–340. [Online]. Available: https://doi.org/10.1007/s11265-015-1056-7

[22] F. Conti and L. Benini, "A ultra-low-energy convolution engine for fast brain-inspired vision in multicore clusters," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 683–688, ISSN: 1558-1101.

[23] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference," vol. 37, no. 11, pp. 2940–2951.

[24] A. Kurth, A. Capotondi, P. Vogel, L. Benini, and A. Marongiu, "HERO: an open-source research platform for HW/SW exploration of heterogeneous manycore systems," in *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems*, ser. ANDARE '18. Association for Computing Machinery, pp. 1–6, event-place: New York, NY, USA. [Online]. Available: https://doi.org/10.1145/3295816.3295821

[25] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for many-core accelerators in heterogeneous embedded SoCs," in *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 45–54.

[26] A. Capotondi and A. Marongiu, "Enabling zero-copy openmp offloading on the pulp many-core accelerator," in *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 68–71. [Online]. Available: https://doi.org/10.1145/3078659.3079071

[27] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar, "High-level language tools for reconfigurable computing," vol. 103, no. 3, pp. 390–408, conference Name: Proceedings of the IEEE.

[28] G. Di Guglielmo, C. Pilato, and L. P. Carloni, "A design methodology for compositional high-level synthesis of communication-centric SoCs," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*. ACM Press, pp. 1–6, event-place: San Francisco, CA, USA. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2593069.2593071

[29] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of FPGA high-level synthesis tools," vol. 35, no. 10, pp. 1591–1604. [Online]. Available: http://ieeexplore.ieee.org/document/7368920/

[30] C. Pilato, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "System-level optimization of accelerator local memory for heterogeneous systems-on-chip," pp. 1–1. [Online]. Available: http://ieeexplore.ieee.org/document/7572091/

[31] J. Choi, S. Brown, and J. Anderson, "Resource and memory management techniques for the high-level synthesis of software threads into parallel FPGA hardware," in *2015 International Conference on Field Programmable Technology (FPT)*, pp. 152–159.

[32] P. Davide Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, "Slow and steady wins the race? a comparison of ultra-low-power RISC-v cores for internet-of-things applications," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 1–8.